2018.2.27

Lecture 11

Depth first search

1. $G = (V, E)$

G: graph

V: vertex $= \{v_1, v_2, v_3, \cdots\}$

$|V| = n$ order of graph

$E = \{(u,v) \mid u, v \in V\}$

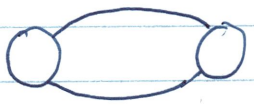
set of edge: ordered pair ⓤ → ⓥ ← directed (u,v) (v,u)

unordered ⓤ → ⓥ ← undirected $(u,v) = (v,u)$

$|E| = m$ size of graph

2.

self-loop ×



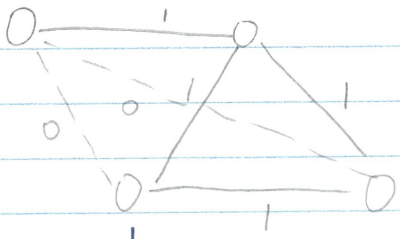
×

simple graphs ⇒ no self loop, no multigraph

✓

3. $G = (V, E)$

binary representation



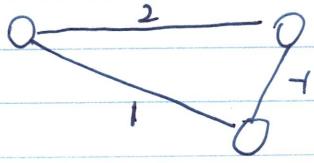
weighted graph:

$G = (V, E, w)$

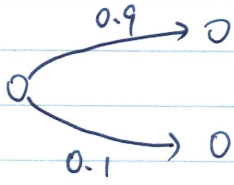
V = vertices

E = edges

$w(e) \in \mathbb{R}$ is the weight on edge $e = (u, v)$

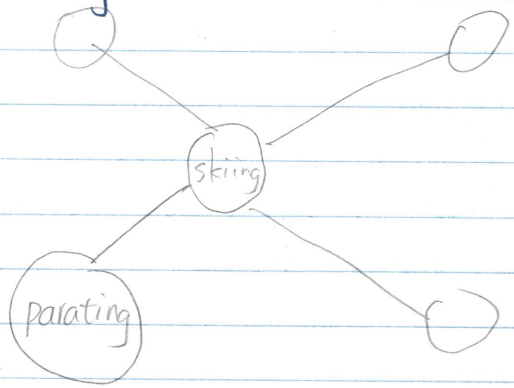


Lecture 11



← probabilities graph

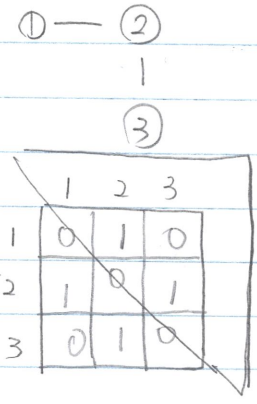
4. labeled vs unlabled
   - ↓ there are "labels" on the verties & edge
     - $G = (V, E, L_V, L_E)$
     - $L_V(v)$ = label for v
     - $L_E(e)$ = label on an edge e



5. Representation
   - Adjecency matrix
   - $G(V, E)$
   - $|V| = n$
   - $|E| = m$

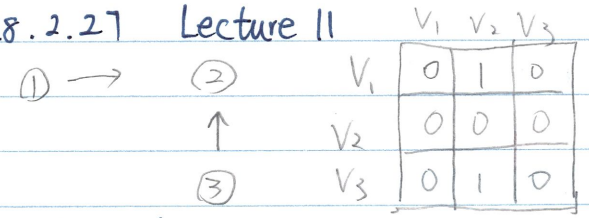
A

	V_1	V_2	...	V_n
V_1	0	1		
V_2	1	0		
⋮			0	
			0	
			0	
V_n				0

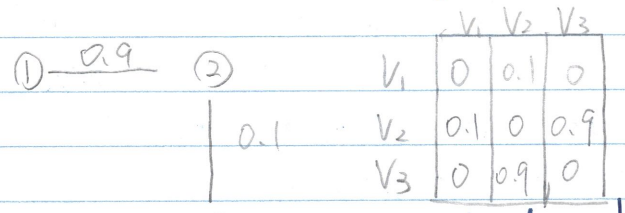


$n \times n$ symmetric binary matrix

2018.2.27 Lecture 11



directed asymmetric matrix



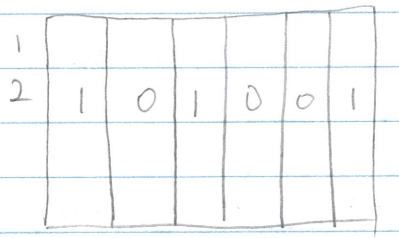
(symmetric ?)

weighted adjcent matrix

Neighbour of a vertex V

$N(v) = \{u \mid (v,u) \in E\}$

↑ set of all vertex "adjacent" to v



$N(2) = \{1, 3, 6\}$

existence of edge $O(1)$

for adjacency matrix $N(v)$ takes $O(n)$ time

A = n x n matrix

$= O(n^2)$ quardratic space

↑ has the record both the existence & absence of an edge.

1). Dense graph

↘ a large fraction of the possible edge exist

G = (V, E)

density = $\frac{|E|}{\#\text{ of possible edge}} = \frac{m}{\binom{n}{2}}$ ← ~~edge~~ # of actual edge ← # of possible edge

undirected = $|V| = n$ $\binom{n}{2} = \frac{n(n-1)}{2}$

2018.2.27 Lecture 11

a graph is dense if $m = \Omega(n^2)$ (edge)

2). Real-world : graphs are sparse

$m = O(n)$

a node / vertex connected to some max const # other nodes

degree(v) = # of nodes that are adjacent to v

$= |N(v)|$ ← size of neighbour

eg: FB n = billion

① degree(u) ≤ 10^6 max

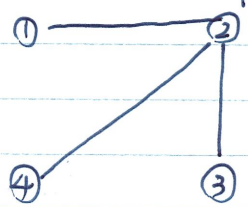
avg degree(n) ≤ 10 - 100

$O(n^2)$ edge ← max

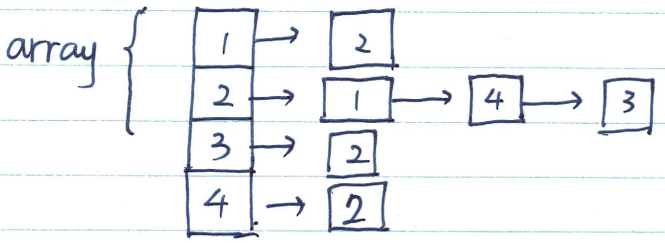
For sparse graph

Adjacency list representation

↳ keep track of presence of an edge.



$|E| = |m|$



Q : (4, 3) exist?

(3, 4)

connect

~~O(1)~~

$O(|N(u)|)$

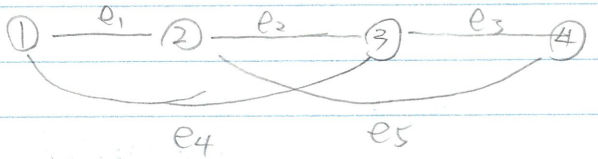
space = $O(m)$ ← degree, edge

↘ O(n) for sparse node

walk versuus path

walk is any sequence of alternating vertices & edge in a graph

2018.2.27 Lecture 11



Walk { ② — e2 — ③ — e4 — ① — e1 — ② — e5 — ④

② — ③ — ① — ② — ④

(2, 3, 1, 2, 4)

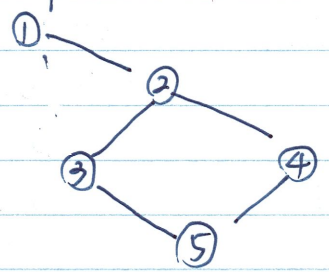
Path: no duplicate vertice allowed, except for the start & end ?

Cycle: is a path with same start & end

① — ② — ③ — ①

③ — ① — ② — ④

DFS: Depth First search



linear time procedure. (basic question: What part of graph are reachable from a given vertex.

DFS explore (u) neighbour

for x ∈ N(u):

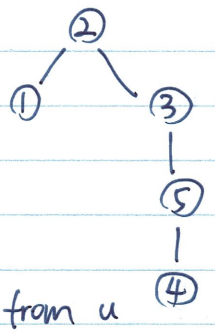
~~DFS explore (x)~~

if not visited(x):

visited(x);

DFS explore(x)

jump to neighbour with smallest index

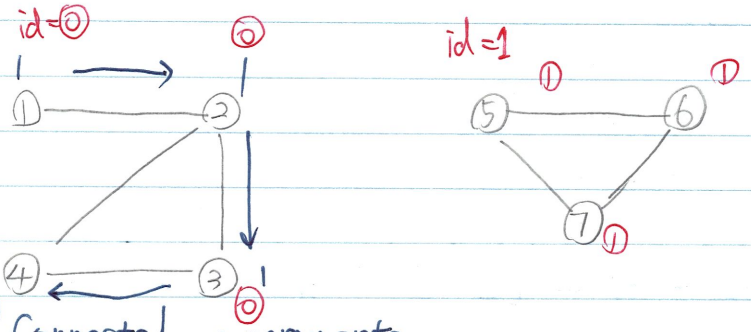


order of exploretion

DFS tree

DFS → find all reachable vertices from u

reachable: ∃ a path for u to s.



Connected components

↘ a maximal set of mutually reachable vertice

$CC_1 = \{1, 2, 3, 4\}$

$CC_2 = \{5, 6, 7\}$

DFS to find connectes components

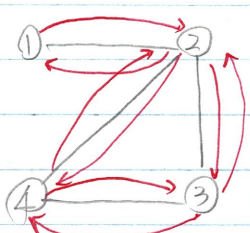
```
CC(G)
    for all u ∈ V          | O(n)
        visited(u) = 0     |
    id = 0
    for all u ∈ V  ← O(n)
        if visited(u) = 0 :
            DFS explore (u, id)
            id = id+1
```

for undirected

# of recursive calls is O(n)

cost of each call ?

```
DFS explore (u, id)
    visited(u) = 1
    id(u) = id
    for all x ∈ N(u)        ← neighbour
        if visited(x) = 0:
            DFS explore (x)
```



every edge will be check exactly twice

O(n) vertice + 2m = O(m)

DFS O(n+m)

O(|V|+|E|)

不管怎么走，都是每个 node 走两遍

for each {x, y} once during explore x, once during explore y

linear time algorithm in the graph "size"

Aggregate analysis : across all nodes

1). each node visites once

2). We look at all its neighbour.

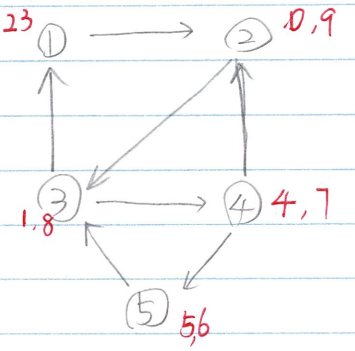
2018.2.27

Lecture 11 Chapter 3

1. DFS with pre & post number

order of visit in DFS exploration

directed graph



② [0, 9]

③ [1, 8]

① [2, 3] ④ [4, 7]

⑤ [5, 6]

DFS pre-order

DFS post-order

u. [x, y]

reachability can be checkes in constant time. after DFS.

Q: is there a path from ② to ① ?

[0, 9] [2, 3]

since [2, 3] < [0, 9]

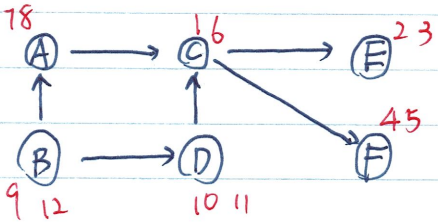
True!

Reachability: source encloses end.

2018.3.6

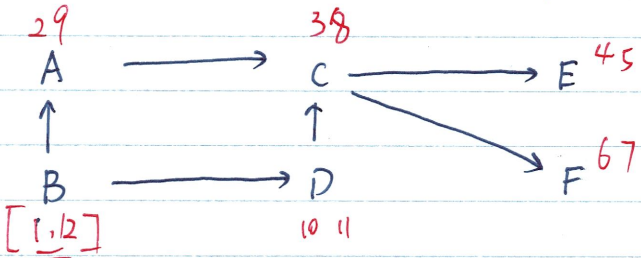
LECTURE 12

1. DFS numbering
   pre-order number
   post-order number



DFS numbering
many of them
1) where we start
2). How we visit the outgoing edges

Containment of ~~interests~~ intervals indicates ancestor-descendent relationship



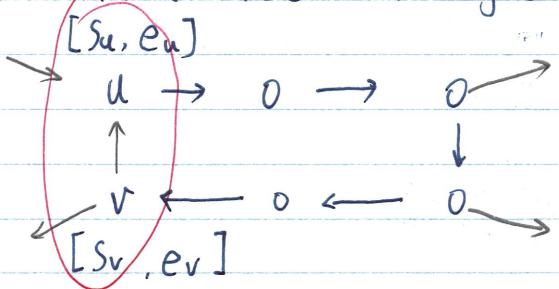
☆ If we start at a source then all vertices reachable from that source will have small interval

2. Detecting cycle

DAG → directed Acyclic Graph

give a directed graph, is it a DAG
→ Does it have a cycle or not ?

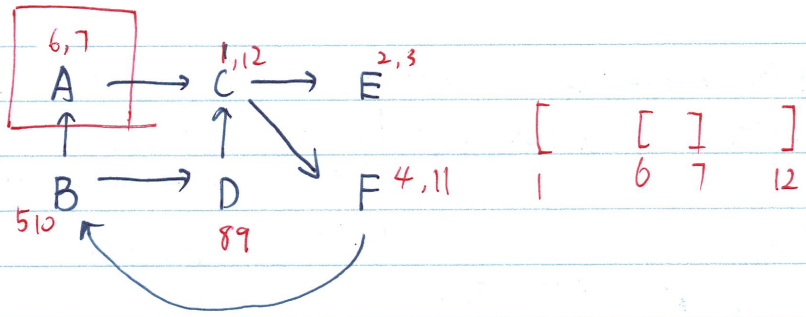
Observation: If there is a cycle, ∃ a node u and v such that



such that: Interval of v ⊆ Interval of u
[Sv, ev] ≤ [Su, eu]

☆ If at v, we found a link to some vertex u such that Intervals of v ≤ Interval of u that has already been visited
then ∃ a cycle

Lecture 12



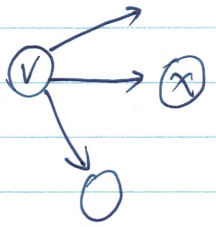
Algo to detect cycles

1) Do DFS intervals — $O(|V|+|E|)$ linear time

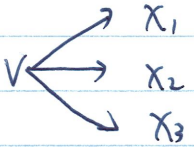
2) ~~$\forall\ v \in V$~~ $O(|V|+|E|)$ ← for all $x \in$ neighbors of V

if interval[x] ⊇ Interval [v]

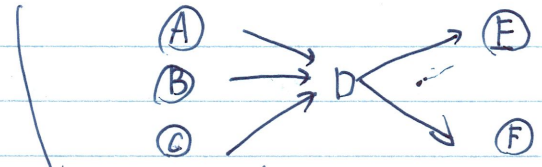
cycle found.



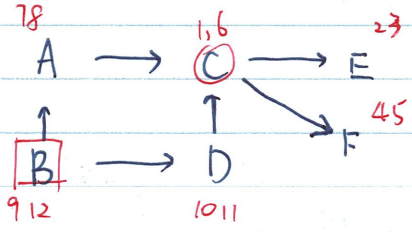
$O(|V|+|E|)$ worst case



3. Topological sort dependency graph DAG



all of the "pre-requisits" of a node must before that node in ordering



only possible for DAG.

Lecture 12

Topo sort:

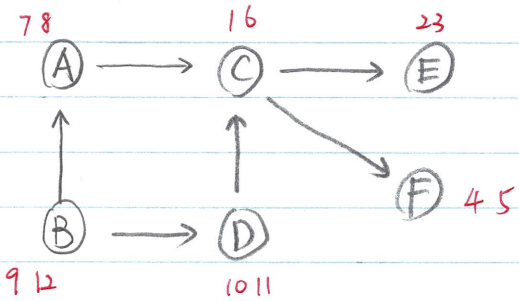
1) Do a DFS numbering $O(|V|+|E|)$
2). ~~Sort~~ output vertice in decreasing order of post [v] value. , order list

~~$|V|\log|V|$~~ $O(|V|)$

This is a topological sort

B D A C F E
12 11 8 6 4 3

$O(|V|+|E| + V\log|V|)$
$O(|E| + |V|\log|V|)$ $O(|V|+|E|)$

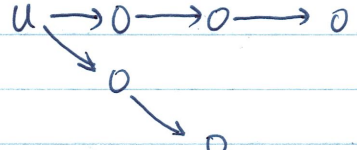


post order list
E - 3
F - 4
C - 6
A - 8
D - 11
B - 12

sorting is for free

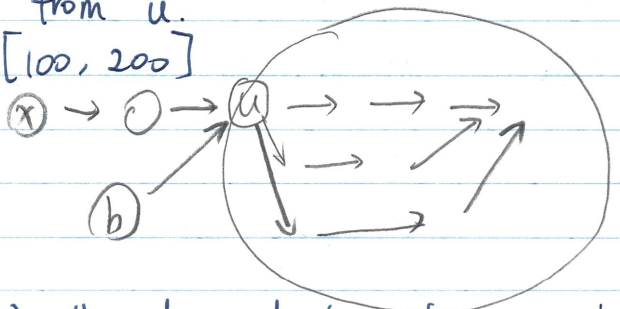
4. Proof of correctness

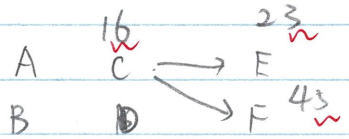
Interval of u [ ]
contains all reachable node's interval



post(u) must be larger than all other nodes that are reachable from u.

[100, 200]





1) all descendants of u have smaller post value.
2). All ancestors of u have larger post value.

Lecture 12. Chapter 3.

3.4 Strong connected components

3. Strongly connected components (SCC)

Directed graph (with cycles)

SCC: maximal set of vertices that are mutually reachable from each other

Connectivity in directed graph: $u \leftrightarrow v$.

$\{x, y\} \in SCC$

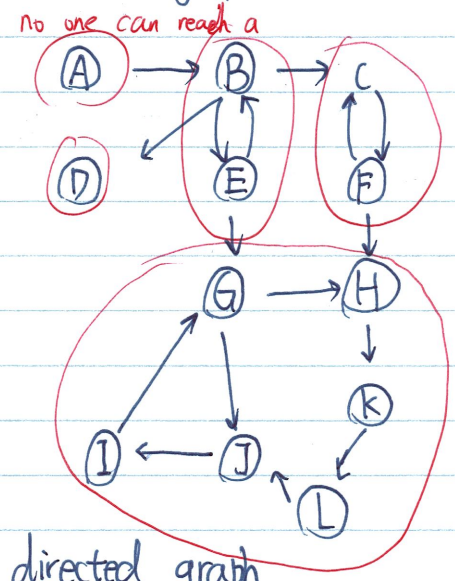
$\Rightarrow$ x can reach y } mutually reachable

y can reach x }

"largest" such set of vertice

↓ we can not add another vertice to SCC.

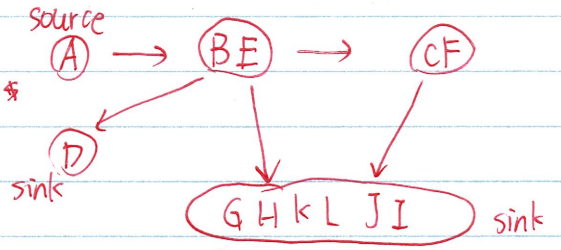
Directed graph has an interrupt DAG structure over scc



directed graph

no one can reach A

D can reach nobody.



SCC DAG

Any directed graph is a DAG over its strongly connected component

SCC Algorithm

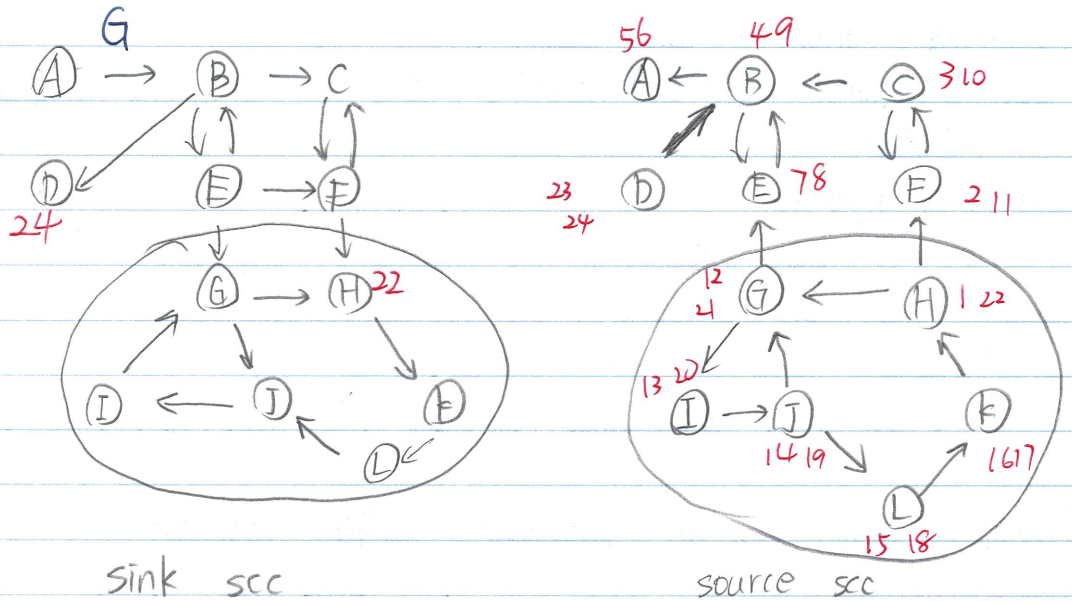
Given a directed graph $G = (V$.

1) Identity successively all "sink" nodes

2). From call sink, do a DFS to extrace the SCC remove it → remove it → mark all vertice as visited

2018.3.6 Lecture 12.

A source vertex in a directed



sink scc

source scc

1) Do DFS numbering as reverse graph

2). Just traverse in decreasing order of post(v) in G and find all reachable vertice from v

every such series a scc

D 24 → D

H → H K L J I G

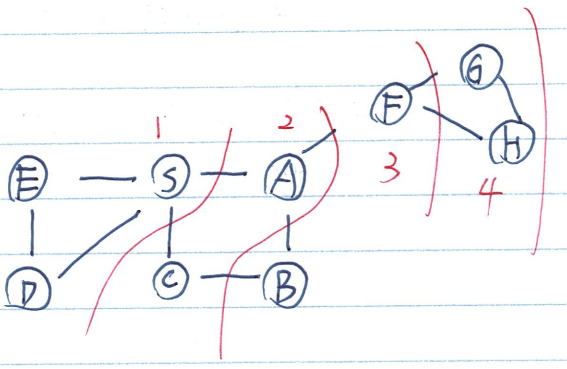
F → F C

2018.3.9

Lecture 13 Chapter 4 Paths in graphs

DFS (chp3)
↘ reachability
BFS
↘ shortest

} $O(|V|+|E|)$



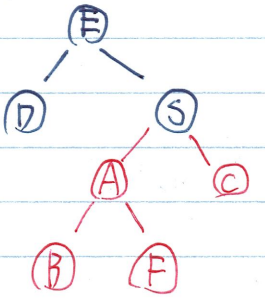
DFS
↘ stack (first in, last out)

E
I
D
S
A
B
C

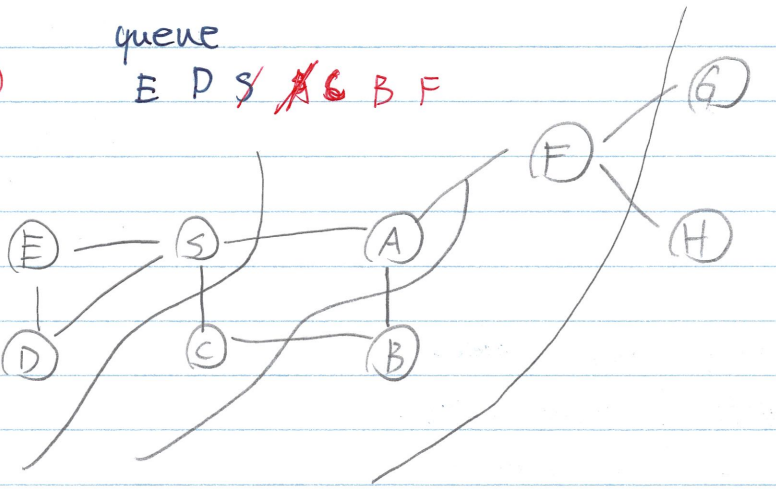
stack: H G F ~~C~~ ~~B~~ A S D E

pop them if we go every we can get

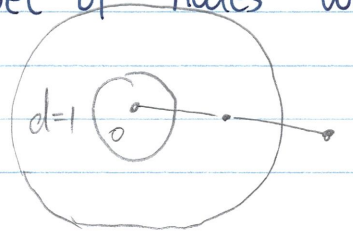
Breadth First search (BFS)
↘ queue (first in first out)



queue
E D S ~~A~~ ~~G~~ B F



R: set of nodes within d hops
↑
region
frontier



d is current distance
current R

unexplored node
dist>d

2018.3.9

Lecture 13

Inductive argument that BFS computes the shortest path from origin to each node

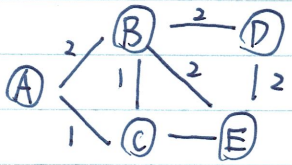
↓ length : # of hops or # of edges on the path

BFS: shortest path for graphs that are unweighted

weight = 1 if edge exists

weight = 0 if edge does not exists

shortest path in weighted graphs



$w(e)$: weight >0

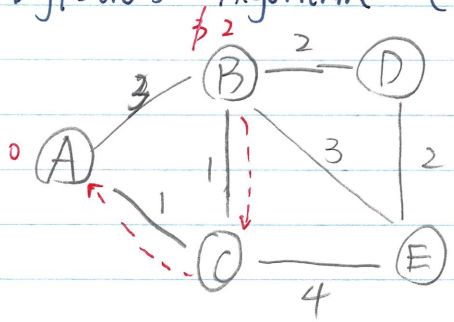
↑

edge positive weight

$\text{weight}(\text{path}) = \sum_{e \in \text{path}} w(e) = \text{cost}(\text{path})$

shortest ~~weightest~~ weighted path from original to all other nodes

Dijkstra's Algorithm (BFS with cost)



Dijkstra's (u)

↑ original

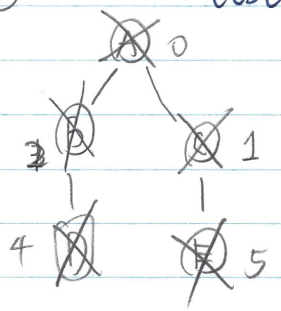
for all $v \in V$

$\text{cost}(v) = \infty$

$\text{cost}(u) = 0$

choose every ~~vertex~~

extract every vertex once from priority queue



2018. 3.9

Lecture 13. Chapter 4 Path in graph

For a given u:

for all neighbors x of u

~~if cost(u) +~~

if cost(u) +

$w(u,x) < cost(x)$:

$cost(x) = cost(u) + w(u,x)$

prev(x) = u

|E| times worst case for updating the cost in the priority queue.

PQ : priority queue

(0). cost of creating the queue

(1). extract min cost vertex

(2). decrease cost operation

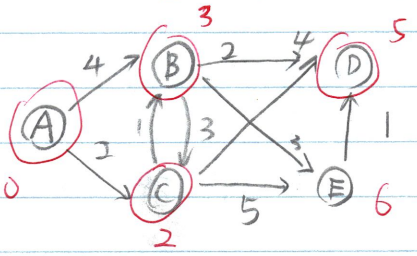
Dijkstra's time complexity

① creation time

② O(|v|) extract min operations

③ O(|E|) decrease cost operations

PQ : Array Implementation (unsorted)



A	B	C	D	E
0	3	2	5	6

dist (A→B) = 3

A→C = 2

A→D = 5

A→E = 6

1) creation time O(v)

2). extract min. scan the whole array O(|v|)

3). decrease operation O(1)

$O(|V|) + O(|V|) \times O(|V|) + O(|E|) \times O(1)$

$= O(|V|^2)$ ← array based PQ.

2018.3.9 Lecture 13 Path in Graph 4.5 PQ implementation

PQ: using binary tree implementation

~~min-heap data structure~~

$O(\log|V|)$ ⇄ extract min, decrease

$O(|V|)$ ← creation.

---

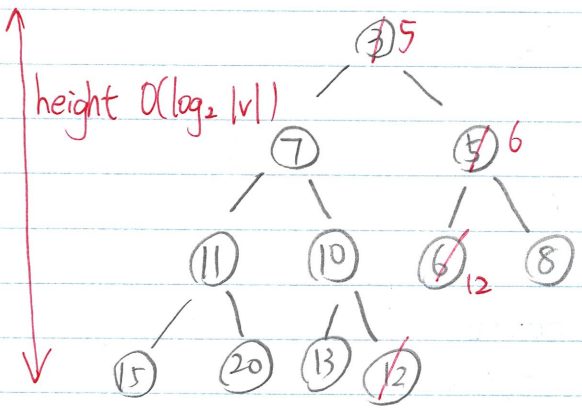
priority ~~tree~~ Queue: binary tree implementation

min-heap data structure

↘ for each node u is in the binary tree

Invariant cost(u) < cost of any of the two children

⇒ root of the tree is min.

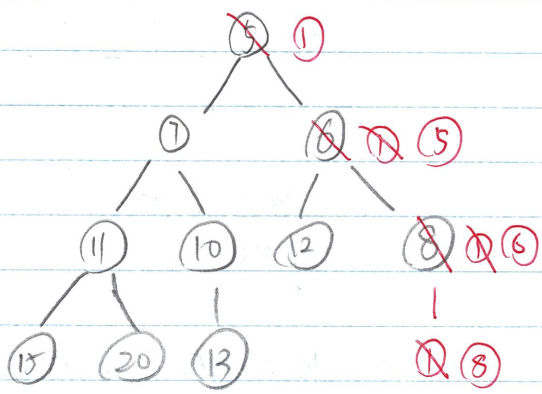


return ③

adjust the min heap.

tree keeps the cost of each node

Decrease cost.



$\log_2 n = \log_2|V|$

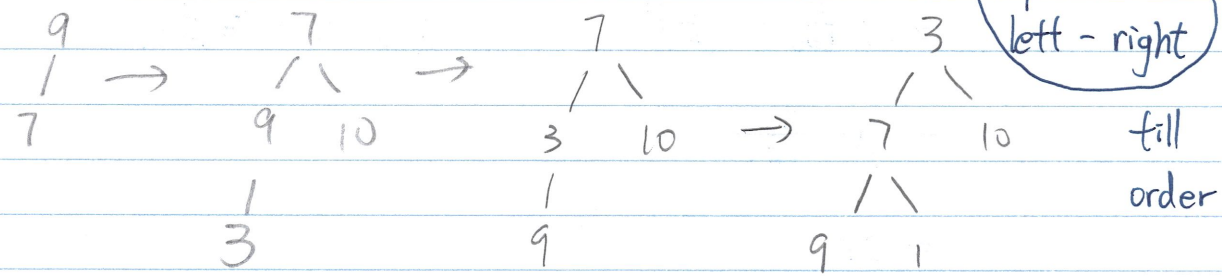
↑

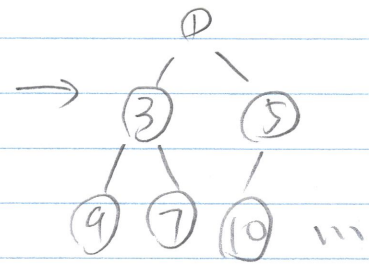
# of node in the PQ

2018.3.9

Lecture 13. Chapter 4.5 PQ Implementation

Create PQ [9 7] 10 3 1 5 4 2 6 8



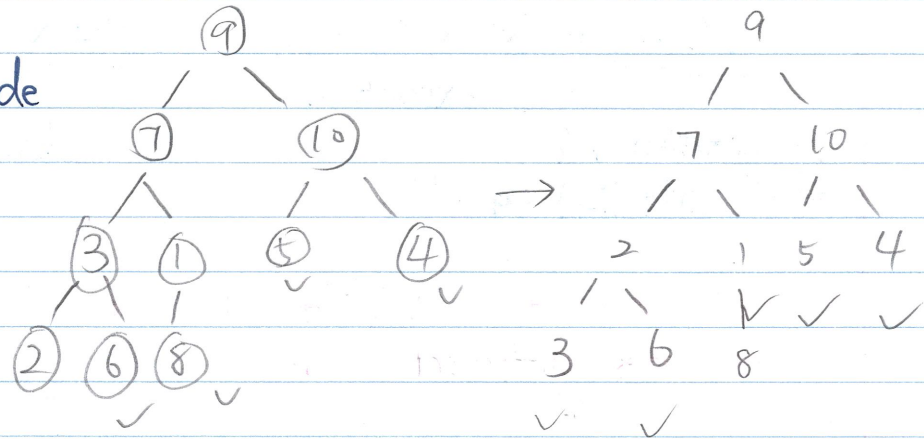


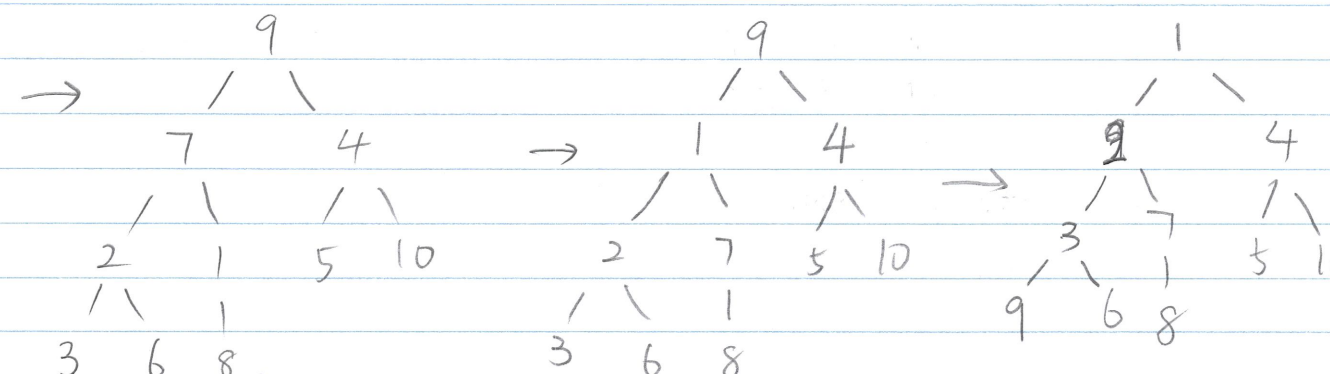
time : $|V| \cdot \log |V|$

↑ "naive"

linear time :

height	# node
3	2^0
2	2^1
1	2^2
0	2^3





Total cost time $= \sum_{n=0}^{\lfloor \log_2 n \rfloor} h \cdot 2^{\log_2 n - h}$

$= \sum_{n=0}^{\lfloor \log_2 n \rfloor} \frac{h \cdot 2^{\lfloor \log_2 n \rfloor}}{2^n}$

Lecture 13

4.5 PQ implementation

2018. 3. 9

$$= n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}$$

$$\leq n \cdot \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h$$

geometric formula

$$\sum_{n=0}^{\infty} h \cdot r^h = \frac{r}{(1-r)^2}$$

$$= n \cdot \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2}$$

$$= 2n$$

$$= O(n) = O(|V|)$$

PQ: min heap:

- → creation : $O(|V|)$
- → decrease / extract - $O(\log |V|)$

Dijkstre :

$$\underbrace{O(|V|)}_{\text{creation PQ}} + |V| \times \underbrace{O(\log |V|)}_{\text{extract min}} + \underbrace{|E| \times O(\log |V|)}_{\text{decrease cost}}$$

$$= O((|V| + |E|) \log |V|)$$

A crucial invariant is that the dist values it maintains are always either overestimates or exactly correct.

2018. 3. 20

Lecture 14 4.5 PQ Implementation

1. Breadth First Search

$G = (V, E)$

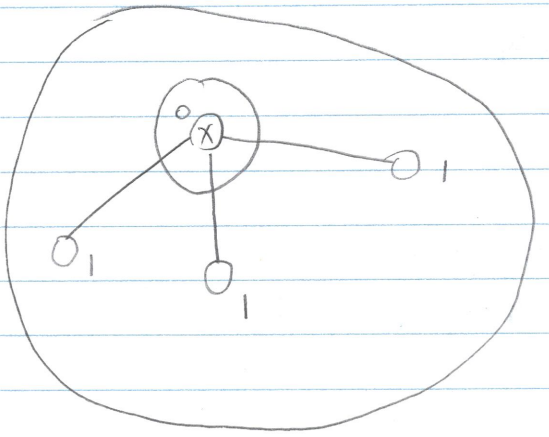
$w(x, y)$ = weight (positive) on the edge (x, y)

"length" = weight ≡ cost of the entire path

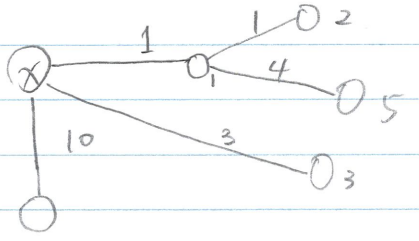
Task: finding shortest paths
- Dijkstra's method (weighted, + ve)
- BFS (0/1) edges

G, source x, find shortest path to all other vertices

BFS:



Dijkstra's algo replacing # of hops with weighted hops



Always pick the current smallest cost vertex and expand that

explored set unexplored set
- ↳ PQ (priority queue)
  - binary tree
    - delete min $\log |V|$
    - Insert / decreasekey $\log |V|$

Lecture 14 4.5 PQ Implementations

2018.3.20

binary tree: { delete min $\log |V|$
insert / decrease key $\log |V|$

total cost: $O((|V|+|E|)\log|V|)$

PQ: array: ↘ delete min $O(|V|)$
↘ insert / decrease $O(1)$

$O(|V|^2) + O(|V|+|E|) = O(|V|^2)$

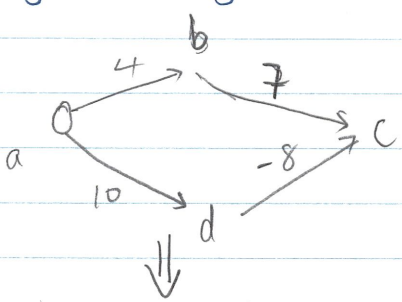
Dijkstra's $O(|V|)$ delete mins
$O(|V|+|E|)$ decrease key

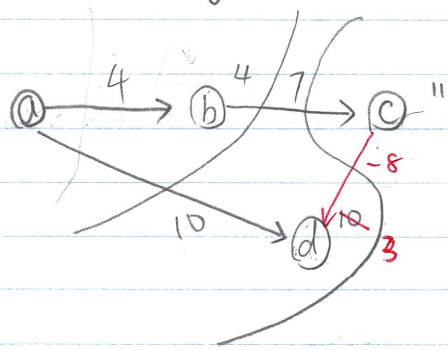
total cost: binary PQ Array PQ.
$|V|\log|V| + |E|\log|V|$ vs $O(|V|^2)$

good for sparse graphs
$|E| = O(|V|)$
$\approx O(|V|\log|V|)$

dense: $E = O(|V|^2)$
$|V|^2\log|V|$ vs $O(|V|^2)$
$|E|$ array is good

2. Negative weights





counter - example to Dijkstra's

Lecture 14

2018.3.20

4.6 Shortest paths in presence of negative edges

3. Bellman - Ford methods.

update the weights of all vertices in ends iteration

$\forall$ vertice $u \in V$

for all neighbours y of u:

update weight:

$$cost(u) + w(u, y) \leq cost(y)$$

$$cost(y) = cost(u) + w(u, y)$$

X → O → O → ... → O

max # of edges in any path from X $\leq |V| - 1$

exactly $|V| - 1$ iteration

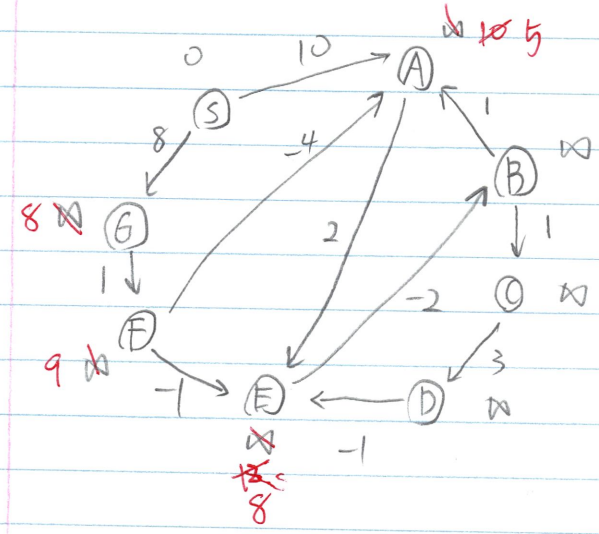
In each iteration, we update all ~~of~~ cost

total cost: $(|V| - 1) \times (|V| + |E|)$

$= O(|V|^2 + |V| \cdot |E|)$

$O(|V| \cdot |E|) \longrightarrow$ sparse $O(|V|^2)$

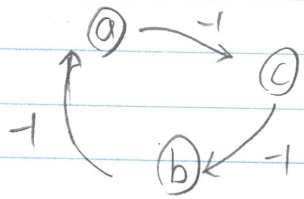
↘ dense $O(|V|^3)$



Iteration:

1

2



1) disallowed negative cycles

2) shortest cost is ill-defined

how to detect this

If we restrict to "paths" Bellman Ford will not work

1) find all possible paths

2018.3.20 Lecture 14 4.6 Shortest Paths in the presence of negative edges

between ⓧ any each ⓨ

source

choose min cost path exponential time method

how to detect cycle (negative)

run bellmon Ford one more iteration, if we still find smaller path, so we can proof it negative cycle.

$O(|V| \cdot |E|)$

Chapter 5 Greedy Algorithm

2018.3.20 5.1 Minimum spanning tree.

1 Greedy Alogrithm

↘ "local" information

→ pick the best available option

Build up solution piece by piece, always choose next piece that offers most obvious and immedi benefit

Minimum ~~tree~~ spanning tree

Iput $G = (V, E)$, with weights

Output: 1) Tree

2). spanning tree

span all the vertex

$T = (V, E')$

$T \subseteq G$ (↑ subgraph)

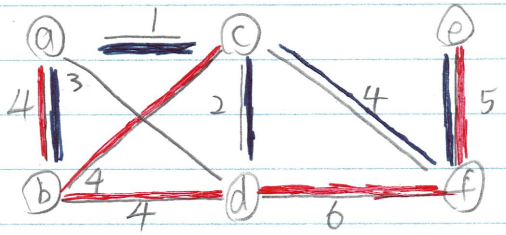
spanning tree: set of vertices in T includes all verties in G

$E' \subseteq E$

3). cost of $T = (V, E')$

$cost(T) = \sum_{e \in E'} w(e)$

find a tree T with smallest positive cost (T)



T_1 is a spanning tree

cost (T) = 23

T_2 is a spanning tree

cost (T_2) = 16

2. Prim's Algorithm (similar to Dijkstra's)

1) sort the edge in increasing order $|E| \log(|E|)$

2). T = { 1st edge in tree }

for each edge in sorted order ← $|E|$

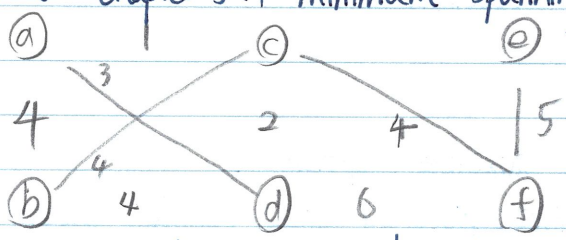
add if and only if one end point is in T

add edge to T if one end point is in T & we do not create a cycle O(1) time

$O(E \log(|E|) + |E|)$

Lecture 14

2018.3.20 Chapter 5.1 Minimum spanning trees



G is undirected

1). ac, cd, ad, ab, bc, bd, cf, ef, df

1 2 3 4 4 4 4 5 6

2).

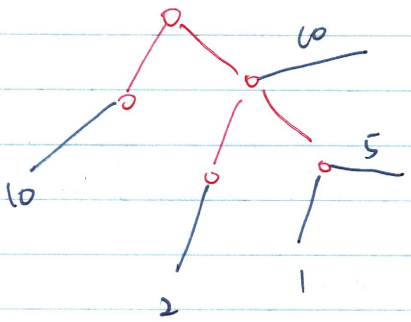
{a, e}
{a, c, d}
{a, c, d, b}
{a, c, d, b, f}
{a, c, d, b, f, e}

cost (T) = 16

1). sort the edge

2). T = { smallest edge }

Iteratively add the least weight edge with one end point not in tree



Greedy: add the least cost edge

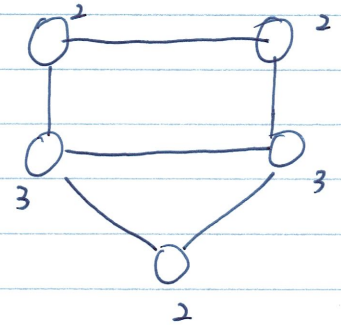
hw 5 Review

2/

(b). matrix → $O(|V|^2)$

For all vertices

3. i) DFS / BFS on G to find all degree → $O(|E|+|V|)$



ii) For $v \in V$

for u adjacent to v:

add 1 to v's node

$\Rightarrow O(|V|+|E|)$

4.

a). Sort topologically $v_1, v_2 \cdots v_n$

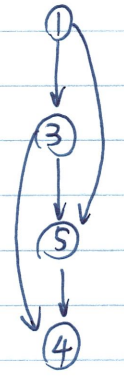
for i is n down to 1

$r(v_i) = w(v_i)$

for $(v_i, u) \in E$.

if $r(u) > r(v_i)$

$r(v_i) = r(u)$



b). Turn G into DAG of SCCs

2018.3.23 Lecture 15 5.1 Minimum Spanning tree

MST: Minimum spanning tree

Given $G = (V, E)$ with weights

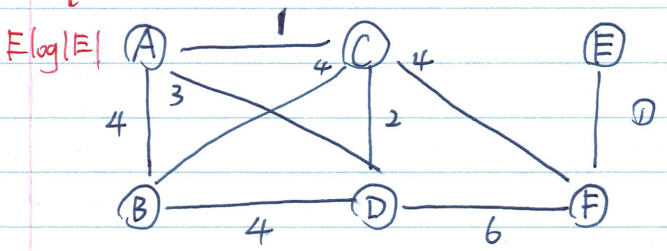
select a min cost tree, $T = (V, E')$ $E' \subseteq E$.

$cost(T) = \sum_{e \in E'} w(e)$

prim's Algorithm

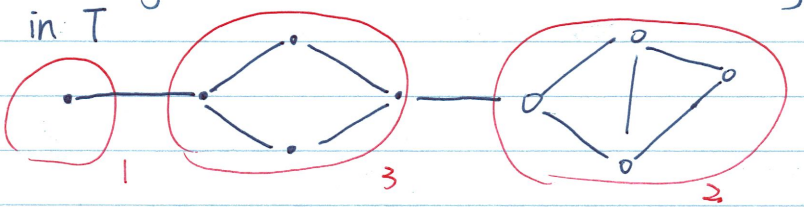
1). Sort edges of G in increasing order of weight

$E \log |E|$



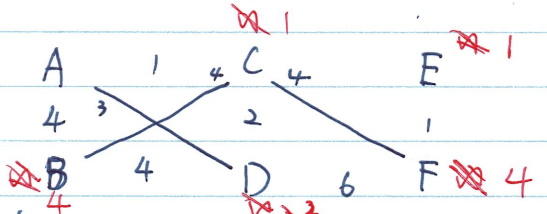
AC	EF	CD	AD	AB	BD	CF	DF
1	1	2	3	4	4	4	6
			↑				
			create cycle				

2). Add edges in sorted order if exactly one end point is in T



Prim's: invariant:

The correct partial MST is always connected



pick a start vertex u_0

$cost(u_0) = 0$, cost of all other vertex $= \infty$

Insert all vertices into a PQ

2018.3.23 Lecture 15 5.1 Minimum spanning tree

```
while PQ is not empty
    u = deletemin(PQ)
    for all neighbors x of u
        if cost(x) > ~~cost(u)~~ cost(u,x)
            cost(x) = w(u,x)
            decreasekey(x, cost(x))
```

O(|V|) for deletemin | binary tree PQ
O(|E|) decrease key

O(log|V|) cost

∴ Total cost: O((|V|+|E|) log |V|)

PQ:
A - 0
* - ∞
↓
B - 4
~~C - 1~~
D - ~~3~~ 2
* - ∞
↓
B-4
D-2
F-4
*-∞

Why does greedy work for MST

cut - property

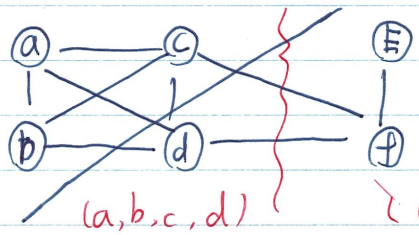
If we have a cut, then we can always choose the smallest edge that cross the cut.

G. graph

$S \subseteq V$: subset of vertics

(S, V-S) is a ~~set~~ cut (or 2-way partition) of G

S (subset) $\bar{S}$ (complement V-S)

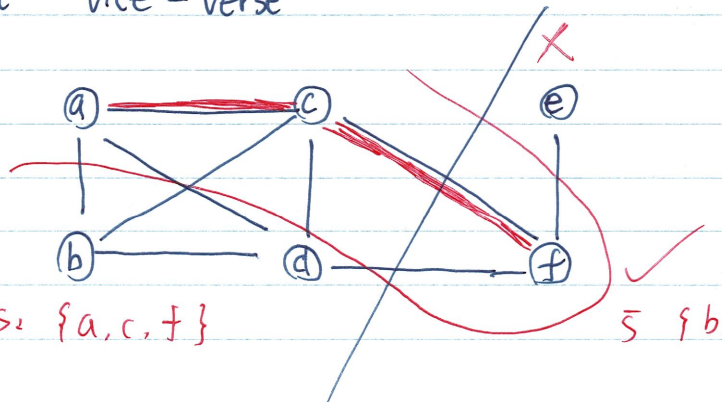


cut (S, V-S)

~~repeats~~ T = (V, E')

respect iff no edge in E' spans or cross from S to V-S or vice-verse

不要把 MST 切断.

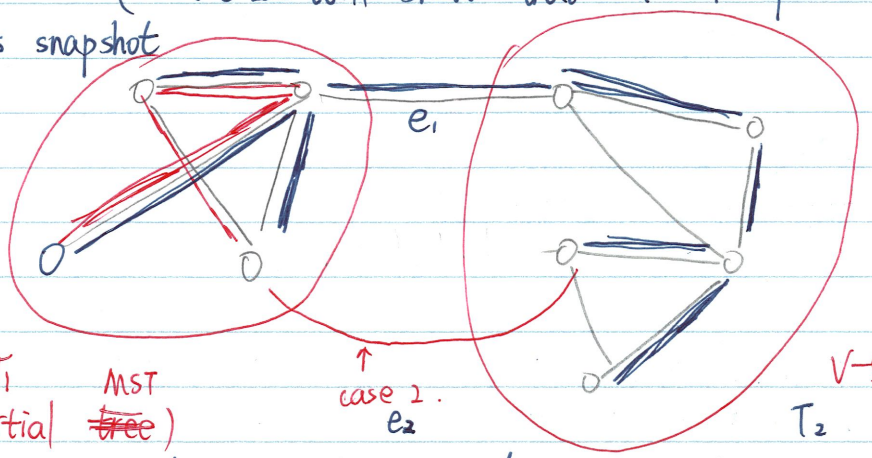


2018. 3. 23. Lecture 15 5.1 Minimum spanning tree.

proof: Given G, and given a partial MST, T

cut-property:
- select a cut that ~~respects~~ T
  - pick the smallest weighted e and add to T
    - $T' = T \cup \{e\}$
- we ~~sh~~ will show that T' is part of some MST

Prim's snapshot



Assume $w(e_1) \leq w(e$

Case 1: there is only one edge cross the ~~tree~~ cut

e_1 is the only one, the smallest as well, e_1 has to be select

Case 2:

Two trees connected by e_2, final MST

If there are ~~two~~ more edges crossing the ~~tree~~ cut

show that selecting the least cost

cost of ~~find~~ final T: $cost(T_1) + cost(T_2) + w(e_2)$

$\exists$ T' that includes e_1

$T' = T_1 \cup \{e_1\} \cup T_2$

$cost(T') = cost(T_1) + cost(T_2) + w(e_1)$

$cost(T') \leq cost(T)$ because $w(e_1) \leq w(e_2)$

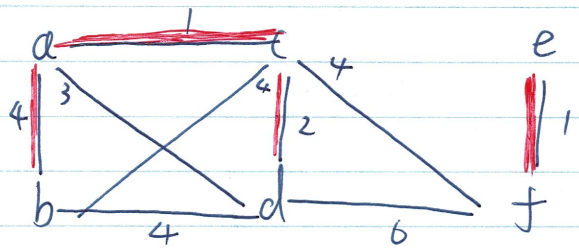
But T is a MST, So it least cost

$cost(T') = cost(T)$

2017-3-23 Lecture 15 5.1 Minimum Spinning tree.

kruskal's algorithm

maintain a "forest": Pick set of edge in increasing order of weights; current "tree" can be disconnected, but there are no cycles



a). Sort:

ac	ef	cd	ad	ab	bc	bd	cf	df
1	1	2	3	4	4	4	4	6
✓	✓	✓	✗	✓	✗	✗	✓	

b). keeping an adding ~~order~~ edges in sorted order

1). check for cycle in the forest each time] DFS on the forest

2). stop where we have a single tree & all vertices have been checked

step a: sorting $O(|E| \log |E|) = O(|E| \log |V|)$ $\quad |E| = O(|V|^2)$, $\log |E| = \log |V|^2 = 2 \log |V|$

step b(1): $O(|E|)$ ← outer for loop

$O(|E| \cdot |V|)$ → detect cycle for each edge in the forest

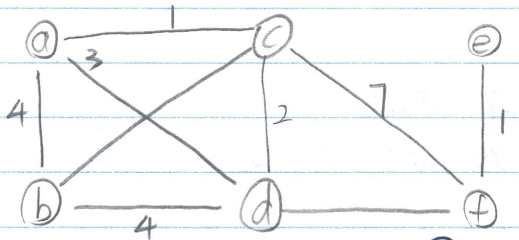
DFS: $O(|V| + |E'|) = O(|V| + |V|) = O(|V|)$

final MST has exactly $|V|-1$ edge & $|V|$ vertice

Cost: $|E| \log |V| + |E| \cdot |V|$

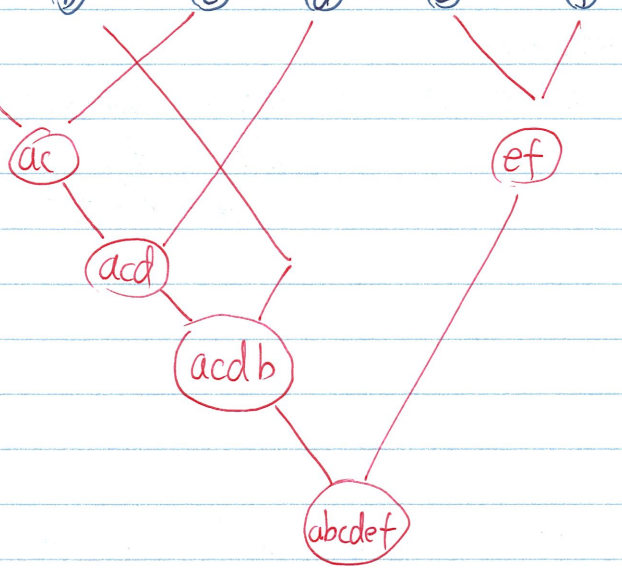
correctness: cut property (given a cut that respects the forest, adding the least edge that cross the cut, always lead to some MST

2017.3.23 Lecture 15 5.1 Minimum spanning tree.



Union-find data structure

1). a b c d e f



cycle can only be with a component

a - d

find operation : are they part of the same component ?
$O(\log |V|)$ time

union operation : $O(1)$ merge 2 component into 1

$E \cdot \log|V| + O(|E| \cdot \log|V|)$

sort — for each edge — find operation

$O(|E| \cdot \log|V|)$ ← Kruskal's time

2018.3.27 Lecture 16 5.1 Minimum Spinning tree.

MST : Minimum Spanning tree

Kruskal's Algorithm

1). sort the edge in increasing order

2). add edge as long as there are no cycle in the partial MST

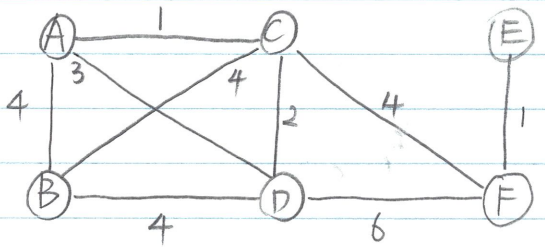
$O(|E| \cdot |V|)$

union-find data structure

Kruskal's maintain a set of connected components

partial tree

forest ← MST



$O(E \log |E|)$ 1) Sort the edge in increase order

2). Create the union-find data structure on V

$O(|V|)$ { A 0, B 0, C 0, D 0, E 0, F rank of the node

$\pi(x)$: parent pointer for x

total : $O(|E| \log |V|)$ → $O(|E| \log |V|)$ → $O(5|E|)$

$r(x)$: rank of x use rank to calculate the tree

x is a root iff $x = \pi(x)$

3). For each edge $e = (x, y)$: in sorted order

if find(x) ≠ find(y) : } ⇒ looking at roots, x, y belong to different component

$O(1)$ $C_{xy} = \text{union}(C_x, C_y)$

remove C_x, C_y

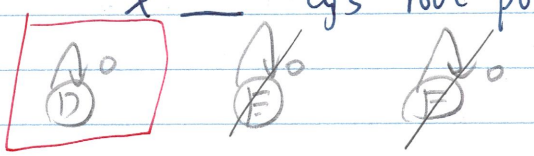
- proportional to the height of tree
- make short tree poin to high tree

2018. 3. 27 Lecture 16

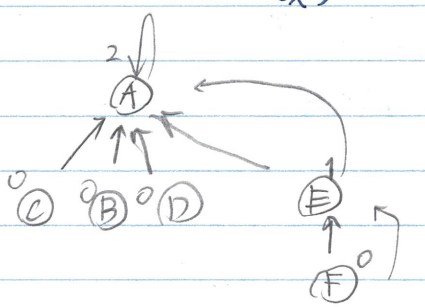
5.1 Minimum spanning tree

union: only look at roots of the two components

(a). If C_x & C_y's roots have the same rank, increase C_x's rank & — C_y's root point to C_x



(b). If C_y's rank is lower, just change the root pointer to C_x's



find(x):

while ($x \neq \pi(x)$)

$x = \pi(x)$

return x

cost of find(x)?

O(log|v|)

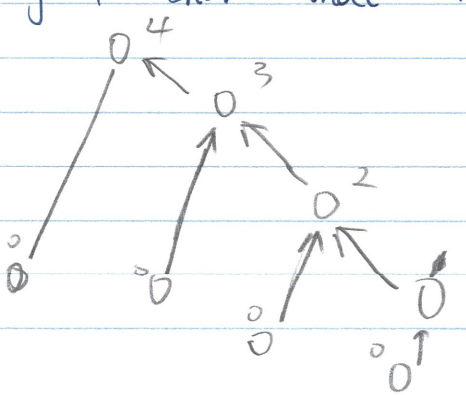
rank of a component determine the running time

find(x) = O(r) time

where r is the rank of C_x

(at most r steps for finding (x)

goal: show that $r = O(\log|v|)$

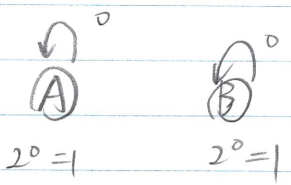


Observation: If rank of a component is r then, there are at least 2^r nodes that component.

2018.3.27 Lecture 16 5.1 Minimum spanning tree

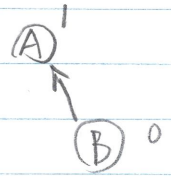
☆ proof by induction

base r=0



$2^0 = 1$ $2^0 = 1$

base r = 1



$\geq 2^1 = 2$

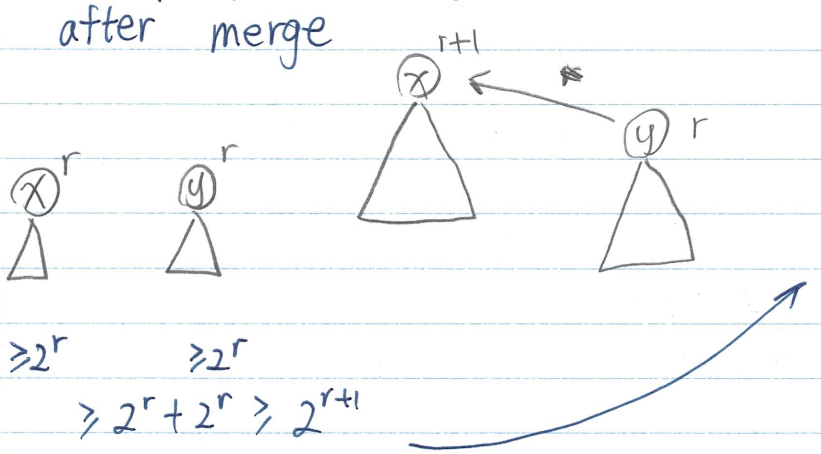
node in that component

Induction: 2 rank r node

x, & y

$|C_x| \geq 2^r$ $|C_y| \geq 2^r$

after merge



$\geq 2^r$ $\geq 2^r$

$\geq 2^r + 2^r \geq 2^{r+1}$

$n \geq 2^r$, rank of final component

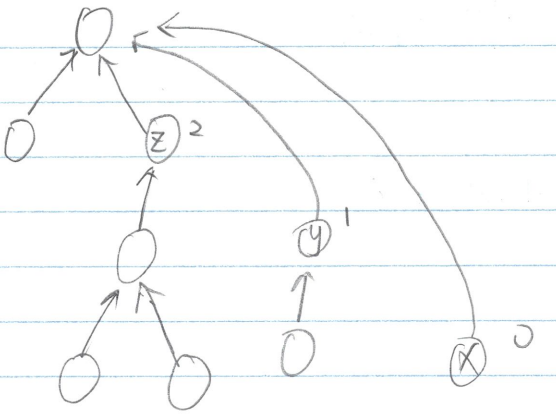
$\log(n) \geq r$

⇒ largest rank to $\log n = \log |V|$

Trick: path Compression

2018.3.27 Lecture 16 5.1 Minimum spanning tree

Trick: path compression



find (x):

we also change the parent pointer for all node on the path to point to the root!

amortized analysis

1). find → $O(\log |V|)$ in worst case

subsequent operations are $O(1)$

↓ overall amortized cost

$O(\log^* |V|)$

# of repeates application of $\log$ before we ~~we~~ got the value 1

$|V| = 2^{65536}$

1) $\log_2 |V| = 65536 = 2^{16}$

2) $\log_2 2^{16} = 16 = 2^4$

3) $\log_2 2^4 = 4 = 2^2$

4) $\log 2^2 = 2 = 2^1$

5)

$\log^* 2^{65536} = 5$

2×|E| find operation

$= O(|E| \cdot \log^* |V|) \simeq O(5|E|)$

↑ practical size

2^{65535}

2018.3.27 Lecture 16 5.1 minimum spanning tree.

Kruskal's & Prim

$|E|\log|E|$ $(E+V)\log V$

$= |E|\log|V|$ $= |E|\log|V|$

$\Rightarrow |E|$

$\therefore$ $|E|$ VS $|V|\log|V|$

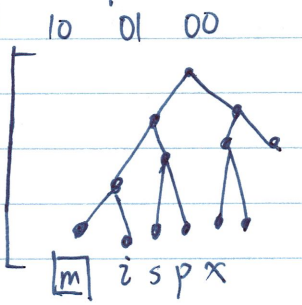
Lecture 16

2018. 3. 27 Bit compression 5.2 Huffman encoding

mississippi

1. Alphabet { m, i, s, p, x }

fixed length encoding



code tree

000 001 010 1010

m i s s

encoding cost(s) = 12 × 3 = 36 bits

decoding : 000 010 001

m s i

2. Variable length coding

code length should be inversely related to frequency

mississippix

m = 1
i = 4
s = 4 ⟹ sort
p = 2
x = 1

m	x	p	s	i	
1	1	2	4	4	← frequency
10	11	01	1	0	← code

bad solution

problem : encoding : m i s s i

10 0 1 1 0 ← insert the encoding length of each "character"

decoding : ?

prefix-free codes : no code for a character should be contained in another as a prefix

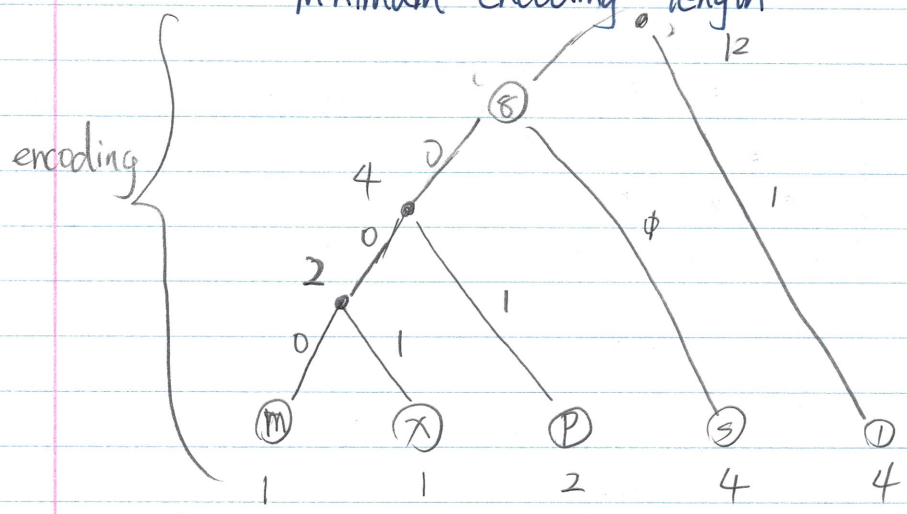
Now, there will be no ambiguity in decoding

2018.3.27 Lecture 16 5.2 Huffman code

Huffman code:

optimal prefix-free code

↓ minimum encoding length



M = 0000	1x4 = 4	
X = 0001		4
P = 001		6
S = 01		8
i = 1		4

encoding: m i s s i s s i p i x

1 1 1

0000 1 01 01 ...

↓

m i s

$\sum_x f_x l_x$ (frequence x length)

encoding length / cost

m = 1x4 = 4

x = 1x4 = 4

p = 2x3 = 6

s = 2x4 = 8

i = 4x1 = 4

fixed code length vs Huffman code

36 26 bit.

2018.3.30 Lecture 17 5.2 Huffman Code

Huffman Coding

variable length

pretix free code

Σ ← Alphabet $\underbrace{|\Sigma| = n}_{n \text{ characters}}$

for each character . $\in \Sigma$

f_i → count → $\frac{f_i}{\sum f_i} = p_i$ ← probability (count)

probability → $\underbrace{f_i}_{\text{probability}} \times N = \text{count}$ (N: string length)

$\underbrace{1, 2, 3, \cdots, n}_{n \text{ set of character}}$

Iteratively merge the two smallest "frequence"

mississippix

m	1
x	1
p	2
s	4
i	4

Given : Σ_{alphater} S ← string to encode

1). Compute f_i for all $i \in \Sigma$

2). Create a PQ (i, f_i) $\forall i \in \Sigma$ $O(n)$

3). While PQ not empty

x = deletemin (PQ) | $O(\log n)$

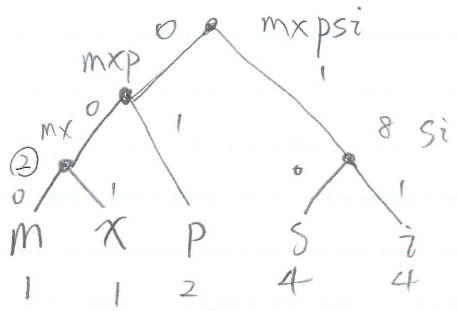
y = deletemin (PQ)

create a new node $z = x \cup y$

$f_z = f_x + f_y$

add (z, f_z) to PQ ← $O(\log n)$ (merge)

n steps, $O(n \log(n))$



Huffman coding $O(n \log n)$ ← if frequence are already given

2018.3.30 Lecture 17 5.2 Huffman encoding

Proof of correctness: There may be ~~serval~~ ~~sew~~ several mincost encoding tree

show that the cost of the Huffman encoding tree

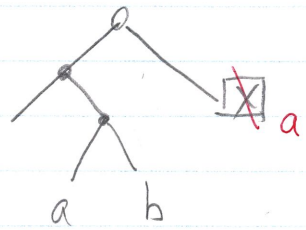
$cost(T) = \sum_{i=1}^{n} f_i l_i$

f_i: frequence, l_i: encoding bit length

where f_i is probability, then cost(T) = expected encoding length

Some encoding tree T

a & b are at the largest



Our tree

x ← deletemin

y ← deletemin

x is the least freq node

y is the second least

$f_x \le f_y$

1) create a new tree T' switch a & x

cost(T') ≤ cost(T)

But T is optimal, eg: mincost

⇒ cost(T') = cost(T)

2). Crete a T'' from T' where we switch y & b

⇒ cost(T'') = cost(T)

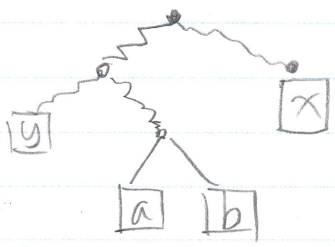
Some optimal tree T

smallest f_x ≤ f_y second

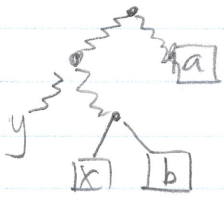
assume that $f_a \le f_b$

$f_x \le f_a$

$f_y \le f_b$



T'



$d_T(y)$: depth of y in tree T

$d_T(x)$, $d_T(a)$, $d_T(b)$

cost(T) − cost(T')

$= d_T(x) \cdot f_x + d_T(a) \cdot f_a - d_T(a) \cdot f_x - d_T(x) \cdot f_a$

2018.3.30 Lecture 17. 5.2 Huffman encoding

$$
\begin{aligned}
&\text{cost}(T) - \text{Cost}(T') \\
&= d_T(x)\cdot f(x) + d_T(a) f_a \\
&\quad - (d_T(x)\cdot f_a + d_T(a)\cdot f_x \\
&= f_x(d_T(x) \pm d_T(a)) + [d_T(a) - d_T(x)] f_a \\
&= (d_T(x) - d_T(a))(f_x - f_a) \\
&= \underbrace{(f_a - f_x)}_{\geq 0}(\underbrace{d_T(a) - d_T(x)}_{\geq 0})
\end{aligned}
$$

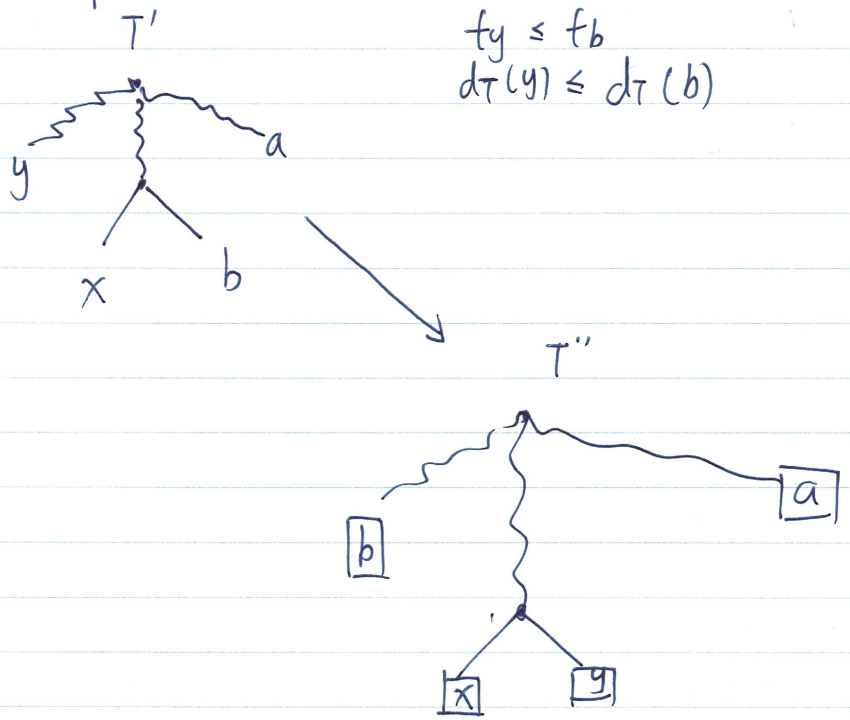
because $f_x \leq f_a$ because $d_T(a) \geq d_P(x)$

$\Rightarrow$ cost(T) - cost(T') ≥ 0

$\Rightarrow$ cost(T) $\geq$ cost(T')

$\Rightarrow$ cost(T') = cost(T) since T is optimal

2nd pair



$f_y \leq f_b$

$d_T(y) \leq d_T(b)$

2018.3.30 Lecture 17 5.4 Set cover

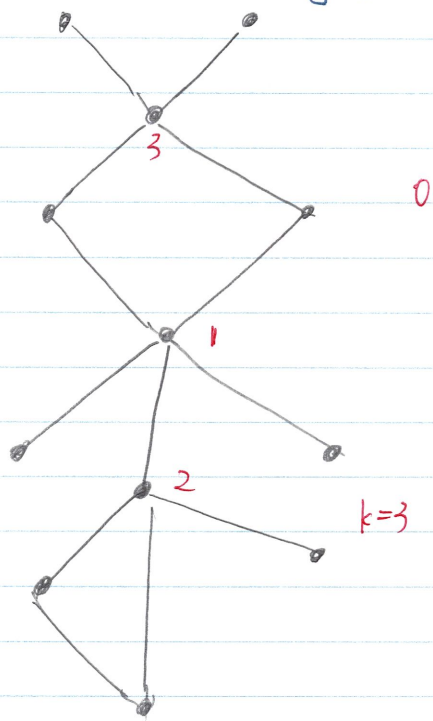
Set cover problem

NP complete ← hard

↘ no polynomial time solution unless P = NP

greedy edge → it gives us an approximate solution that is close to optimal

Assume we have a graph.



every vertex is a city

x is connected to y if dist(x, y) ≤ 30 miles

optimization:

choose that least ~~of~~ # of node / cities to build a hospital.

Sort the vertice by decreasing degree, select one at a time until all nodes are "covered."

k is optimal value

Is k = 3 optimal

Input: G = (V, E)

N_x = neighborhead of x (all neighbors of x)

Problem statement:

Find the optimal number k of nodes

$\{1, 2, \dots, k\}$

such that $\bigcup_{i=1}^{k} N_i = V$

Greedy is not optimal

But the greedy solutions is $O(\log n)$ factor away from optimal

$\Rightarrow$ if the optimal solution is k

then the greedy solution value is $O(k \log n)$

if k is optimal value

$\{1, 2, 3, \cdots, k\}$

total vertices in G is $|V| = n$

observation : $\exists$ some vertex v in σ that cover at least $\frac{n}{k}$ other vertices

proof : assume that for all $x \in \sigma$

$|N_x| < \frac{n}{k}$

$$\sum_{x \in \sigma} |N_x| < \underbrace{\frac{n}{k} + \frac{n}{k} + \cdots + \frac{n}{k}}_{k} < k \cdot \frac{n}{k}$$

contradiction

n_t : # of uncovered vertice after t steps.

$n_0 = n$ ← initially every node is uncovered

$n_1 \leq n - \frac{n}{k}$ ← size of the neighbor of the highest degree node

$n_1 \leq n(1 - \frac{1}{k})$

n_1 nodes remaining, $k-1$ choice remaining at least highest degree

$$n_2 \leq n_1 - \frac{n_1}{k} = n_1(1 - \frac{1}{k}) \leq n(1 - \frac{1}{k})(1 - \frac{1}{k})$$

$$n_2 \leq n(1 - \frac{1}{k})^2$$

$\vdots$

$$n_t \leq n(1 - \frac{1}{k})^t \leq n(e^{-\frac{1}{k}})^t$$ fact : $(1 - \frac{1}{k})^t \leq e^{-\frac{t}{k}}$

the solution is $t = k \log n$

$$n(e^{-\frac{1}{k}})^{k \log n}$$

$$n e^{-\log n} = \frac{n}{n} = 1$$

greedy solution takes at most $t = k \log n$ steps $\equiv$ # of node selected

label_new

label

base case:

$$l_{ij}^{(1)} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{if } (i,j) \notin E \end{cases}$$

Pseudo code:

Initialize via base case

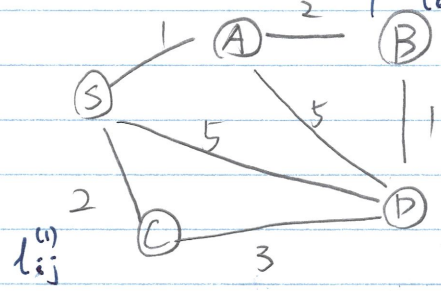
$O(v^3)$ for $m = 2$ to $|v|-1$

for $i = 1 \dots |v|$

for $j = 1 \dots |v|$

$O(v)$ $l_{ij}^{(m)} = \min_{(k,j) \in E} \{ l_{ik}^{(m-1)} + w(k,j) \}$

l_{ij}



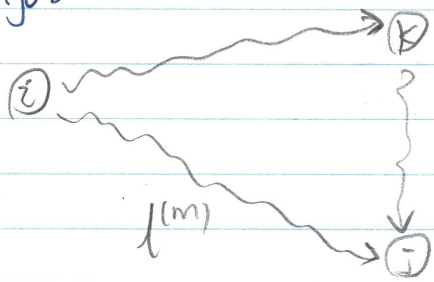
$l_{ij}^{(1)}$

edge	1	2	3	4
AB	2	2		
AC	∞			
AD	5			
AS	1			
BC	∞			
BD	1			
BS	∞	3		
CD				
CS				
DS				

$$l_{BS}^{(2)} = \begin{cases} l_{AS}^{(1)} + w(A,B) \\ l^{(1)} + w(D,D) \end{cases}$$

Alg3:

Algo 3:



$$l_{ij}^{(m)} = \min_{1 \le k \le |V|} \{ l_{ik}^{(m/2)} + l_{kj}^{(m/2)} \}$$

base case:

$$l_{ij}^{(1)} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{~~converse~~} \\ 0 & \end{cases}$$

$m = 1$, initialize $l_{ij}^{(1)}$ $\forall ij$

$O(\log|V|)$ while $m < |V|-1$

  $O(|V|^3)$ for $i = 1 \cdots |V|$:

    for $j = 1 \cdots |V|$

      $l^{(2m)} = \min_{1 \le k \le |V|} \{ l_{ik}^{(m)} + l_{kj}^{(m)} \}$

~~$m = 2m$~~

$m = 2m$

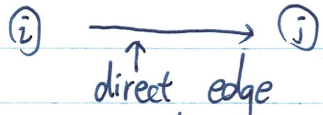
$O(|V|^3 \log|V|)$

#

Floyd-Warshall method $O(|V|^3)$ time algo.

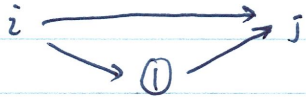
1). Prev approach l_{ij}

restrict the set of vertices that can be on the path

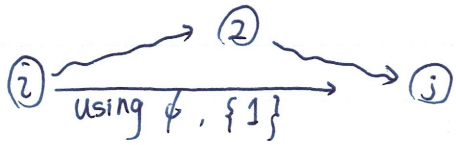


go directly from i to j using

{1}



$\{1, 2\}$



$d(i, j, k) \equiv$ shortest path from i to j using intermediate vertice in the set $\{1, 2, \cdots, k\}$

$= \min \{$

$$d(i, j, k) = \min \begin{cases} d(i, j, k-1) \\ d(i, k, k-1) + d(i, j, k-1) \end{cases}$$

Pseudo code

Initialize $d(i, j, 0) = \begin{cases} w(i, j) \text{ if } (i, j) \in E \\ \infty \text{ otherwise} \end{cases}$

$O(|V|^3)$

```
for k = 1 ... |V|
    for i = 1 ... |V|
        for j = 1 ... |V|
            d(i, j, k) = min { d(i, j, k-1) ;
                               d(i, k, k-1) + d(k, j, k-1)
```

Floyd warshall $O(|V|^3)$ time

Exam II

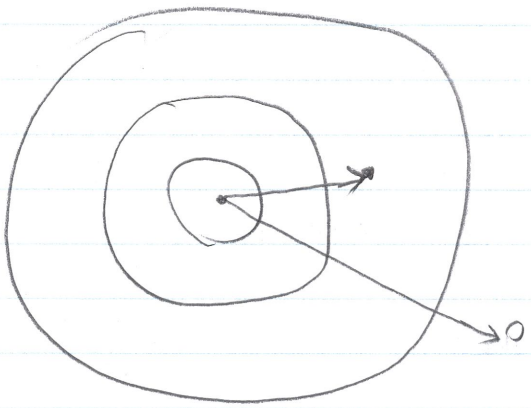
Chapter 3:

- pre-post:
  - ↳ cycle, topo-sort
  - ↳ strong connect component
- SCC

  for any directed graph

  1) SCC's
  2). DAG over the SCC

Chapter 4: BFS



PQ

Dijkstra's

negative-weight

negative cycles cost of decreasing, remove. insert. (array, tree)

Chapter 5

spanning-tree

↳ cost property → greedy strategy → Prim's

↳ Kruskal's → union-find

Huffman codes: PQ.

set cover → greedy strategy

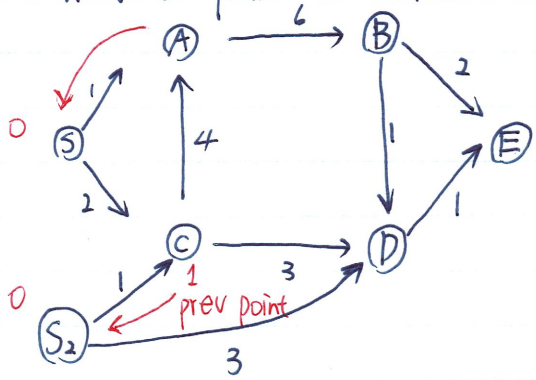
choose the largest degree among the uncove

Dynamic Programming

Reduce the problem size by a small fraction typically, there will be served subproblems

combined result → operations : min
max

Shortest paths in DAG



What's the shortest path to B from a source vertex in the DAG

$d(B)$ $d(B)$ distance to B from a source

Ⓑ

$d(B) = d(A) + \text{weight}(A, B)$

$d(A) = ?$

$d(A) = \min \{ d(s) + w(s, A) , d(c) + w(c, A) \}$

$d(v) = \min_{(x,v) \in E} \{ d(x) + w(x, v) \}$

all incomes edges into V

base case: $d(x) = 0$ iff x is a source vertex in the DAG

(1) recursive formula's defined "backwards" & smaller promise

(2) Computation proceeds in the ~~"forward"~~ "forword" direction from smaller to larger

$O(|E| + |V|)$

All pairs shortest Paths $|V|$ vertices

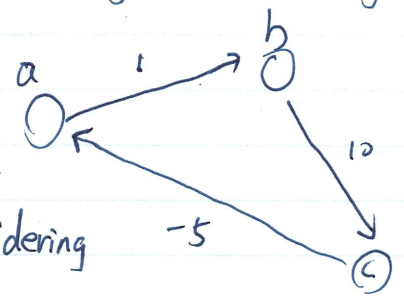
find $d(x, y)$ $\forall$ $x, y \in |V|$

$\Omega(|V|^2)$ lower bond on the cost

G is some Graph, allow negative weights

negative cycle are not allowed

↘ ~~some~~ sum of the weights on the edges in the cycle is < 0



Bellman-Ford

$|V|-1$ steps

start at sources, $d(s,s)=0$,

$d(s,x)=\infty$, update all distance considering each edge in each step.

$O(|V|\cdot|E|)$

Alg 1 APSP: run Bellman-Ford $|V|$ times with each vertex as the starting vertex.

$|V| \times O(|V|\cdot|E|)$

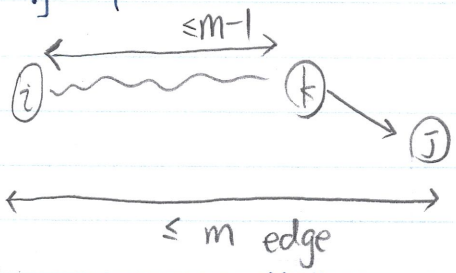
$O(|V|^3\cdot|E|)$ → spare: $|E| = O(|V|)$, $O(|V|^3)$

→ dense: $|E| = O(|V|^2)$, $O(|V|^4)$

Alg2 APSP (Dynamic programming)

$l_{ij}^{(m)}$ = shortest path from i to j that uses at most m edge

$l_{ij}^{(m)} = \{$



$$l_{ij}^{(m)} = \min_{\substack{1\le k\le n \\ (k,j)\in E}} \{ l_{ik}^{(m-1)} + w(k,j) \}$$

$Prev_{ij}^{(m)}$ = arg min

which predecessor of j gives the min value

C

Exam 2 Review

1. DFS: $O(|V| + 2|E|)$ reachability

   DFS to find connected components:

   $O(|V| + 2|E|)$

2. DAG: directed Acyclie graph.

   no cycle graph.

   Alg to detect the cycle:

   $O(|V| + |E|$

3. Lecture 12. Topological sort

   Do DFS numbering

   output vertice in decreasing order

   $O(|V| + |E|)$

4. Lecture 12 Find a source in a directed graph.

   ① reverse the graph

   ② do DFS numbering

   ③ Traverse in desreasing order of post (v) in Graph, find all reachable vertices from V

5. Lecture 13

   BFS: shortest path for graph that are unweighted

   $O(|V| + |E|)$

   Dijkstra's Algorithm. (BFS with cost)

   $|E|$ x worst case for updating the cost in PQ.

   $O(|V|) + O(\log|V|) \times |V| + |E| \times \log(|V|)$

   creation extract min decrease cost.

   $= O((|V| + |E|) \log|V|)$ Binary PQ Good for sparse graphs

   $O(|V|^2)$ Array PQ Good for dense graphs.

   每条 edge 都走一遍，从 PQ 中删掉后，就表示定性为最小

   Dijkstra can not handle negative path.

Exam 2 Review

6. Lecture 14.
Bellman - Ford methods.
cost: $(|V|-1) \times (|V|+|E|)$
$= O(|V| \cdot |E|)$
Can not handle negative cycles

7. Lecture 14 Chapter 5
Prim's Algorithm, span minimum tree.
$O(E \log E + E)$
↑ sort edge ↑ go through each edge.

8. Lecture 15 Chapter 5
I Kruskal's algorithm.
- pick set of edge in increasing order of weights, "current tree" can be disconnected, but there is no cycles.

a. sorting $O(E \log |E|)$
b. adding edge
① check for cycle each time (DFS) $O(|E|+|V|)$
② stop when we have a single tree.

for outer loop $|E|$ times
∴ cost $|E| \log |V| + E \cdot (E' + V)$
$= O(E \log |V| + E \cdot V$

II. union-find data structure
$O(E \cdot \log |V| + |E| \cdot \log |V|)$
sort ↑ foreach edge ↑ find cycle operation relative with rank

III path compression.
$O(|E| \cdot \log^* |V|) \approx O(5|E|)$